Security issues in the Android cross-layer architecture

Alessandro Armando
University of Genova
Via all'Opera Pia, 13
IT-16145,Genova, Italy
armando@dist.unige.it

Alessio Merlo*
E-Campus University
Via Isimbardi, 10
IT-22060, Novedrate, Italy
and University of Genova, Italy
alessio.merlo@uniecampus.it

Luca Verderame
University of Genova
Via all'Opera Pia, 13
IT-16145,Genova, Italy
luca.verderame@ai-lab.it

ABSTRACT

The security of Android has been recently challenged by the discovery of a number of vulnerabilities involving different layers of the Android stack. We argue that such vulnerabilities are largely related to the interplay among layers composing the Android stack. Thus, we also argue that such interplay has been underestimated from a security point-of-view and a systematic analysis of the Android interplay has not been carried out yet. To this aim, in this paper we provide a simple model of the Android cross-layer interactions based on the concept of flow, as a basis for analyzing the Android interplay. In particular, our model allows us to reason about the security implications associated with the cross-layer interactions in Android, including a recently discovered vulnerability that allows a malicious application to make Android devices totally unresponsive. We used the proposed model to carry out an empirical assessment of some flows within the Android cross-layered architecture. Our experiments indicate that little control is exercised by the Android Security Framework (ASF) over cross-layer interactions in Android. In particular, we observed that the ASF lacks in discriminating the originator of a flow and sensitive security issues arise between the Android stack and the Linux kernel, thereby indicating that the attack surface of the Android platform is wider than expected.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Security and Protection; C.1.3 [**Computer Systems Organization**]: Other Architecture Styles—*cellular architecture*

General Terms

Android Security, Zygote Vulnerability, Cross-layer architecture

1. INTRODUCTION

Android is the most widely deployed operating system for smartphones and recent estimates [11] indicate that it will continue to remain so in next years. Android is a Java stack built on top of a native Linux kernel. Services and functionalities are achieved through the interplay of components living at different layers of the operating system. Security in Android is granted by a set of cross-layers security mechanisms that collectively constitute the Android Security Framework (ASF). The security offered by the ASF has been recently challenged by the discovery of a number of vulnerabilities involving different layers of the Android stack (see, e.g., [3, 7, 6]).

In this paper we argue that a systematic analysis of the interplay among the different layers of Android is necessary. To this aim, we provide a simple model of the interaction among the components based on the concept of *flow*. Our model allows us to reason about the security implications associated with the cross-layer interactions in Android, including a recently discovered vulnerability [3] that allows a malicious application to force the system to fork an unbounded number of processes thereby making the device totally unresponsive. The problem is due to the fact that the invocation of a critical functionality offered by the Zygote process (namely the forking of a new process) is not restricted to the ASF by can be invoked by any application (including malicious ones).

An interesting question is whether the problem is limited to the Zygote process of if instead it is a more general issue in Android. To ascertain this, we have defined and carried out an empirical assessment of the allowed flows within the Android cross-layered architecture. Our experiments indicate that little control is exercised among the Android and the Linux layers, thereby indicating that the attack surface of the Android platform is wider than expected.

The rest of the paper is organized as follows: Sect. 2 introduces the cross-layered architecture of Android; Sect. 3 presents the notion of flow; Sect. 4 illustrates the Zygote vulnerability [3] and the associated malicious flow; Sect. 5 presents our experimental setup and results; in Sect. 6 we discuss the related work and we conclude in Sect. 7 with some final remarks.

2. THE ANDROID CROSS-LAYERED ARCHITECTURE

Android is organized into five layers: Application, Application Framework, Application Runtime, Libraries and the Linux kernel. The top four layers belong to the Android stack while the lower one is a native Linux kernel.

1. *Application Layer* (A). It includes both pre-installed (browser, email, ...) and Java applications installed by the user. Applications are made of *components* corresponding to independent, yet mutually interacting execution modules. There exist four kinds of components: 1) *Activity*, representing a single application screen with a user interface, 2) *Service*, running in the background without interaction with the user, 3)

*Corresponding author

Content Provider, managing application data shared among components of (possibly) distinct applications, and 4) *Broadcast Receiver* responding to system-wide broadcast announcements coming both from other components and the system.

2. *Application Framework* (*AF*). It provides the main services of the platform which are exposed to the applications through API. It contains the *System Server* which is made of modules (i.e. the *Activity Manager Service* and the *Package Manager Service*) which are responsible of the proper management of the Android platform. This layer also includes services for managing the device and interacting with the underlying Linux drivers (e.g. the *Telephony Manager Service* and the *Location Manager Service*).

3. *Android Runtime* (*AR*). This layer contains the Java core libraries and the Dalvik Virtual Machine (DVM), i.e. the runtime core component of Android that executes applications. All requests invoked by upper layers and targeted to lower ones pass through the DVM.

4. *Libraries* (*Lib*). This layer contains a set of C/C++ libraries providing useful functionalities to the upper layers and for accessing data stored on the device. Libraries are widely used by the Application Framework services. For instance, the *bionic libc* provide supports support for performing system calls to the Linux Kernel and *SQL Lite* as provides the functionalities of a relational DBMS.

5. *Linux kernel* (*K*). Android relies on a Linux kernel for core system services. Such services include (1) process management, (2) drivers for accessing physical resources, and (3) support to Inter-Process Communication (IPC). Each Android applications are hosted in a Linux process at this layer. Moreover each Android application is assigned a specific Linux user. In this way the standard access control model of Linux ensures that applications are isolated, i.e. *sandboxing*.
   Physical resources are accessed by means of drivers that are triggered through *system calls*. Applications are not expected to invoke system calls directly; on the contrary, proper services at the AF and L layers are in charge of invoking system calls and provide the needed services to applications.
   IPC is carried out through a) an *ad hoc* driver named *Binder* and b) Unix native *sockets*. The Binder driver allows an application to communicate with other applications or Android services by means of message abstractions called *intents*; sockets are files at kernel level where upper layer components can put data in order to share them with others. For security reasons, communication through sockets is discouraged, albeit not strictly forbidden.

Operations in Android are carried out through interactions among layers in the Android stack. Such interactions constitute the interplay of Android and are implemented through different kinds of calls involving distinct subsets of layers and libraries.
In the following, we provide a simple model of Android interplay sufficient for reasoning about its security implications.

3. CALLS AND FLOWS

Given an Android layer $\ell \in \{\mathrm{A}, \mathrm{AF}, \mathrm{AR}, \mathrm{L}, \mathrm{K}\}$, we indicate with X_ℓ a component/module in the ℓ level. For instance, AM_{AF} indicates the Activity Manager Service in the Application Framework. From now on, we refer to each element able to communicate with others with the term *component*. Interactions among components are carried out through a variety of calls. According to the official documentation [2], the following types of calls are available in Android:

- *Binder call*, binder(obj). It is a call to the Binder driver in the Linux kernel that allows inter-component communications. A Binder call can be invoked by all layers belonging to the Android stack (namely A, AF, AR, and L) and it is directed to the Binder driver in the Kernel. Then, the Binder driver establishes a communication with the target component and delivers the serializable object *obj* to the destination. A Binder invocation is made though an ad-hoc system call, namely `ioctl()`. For example, an application requesting a `startActivity` to the Activity Manager Service will use a binder(`startActivity`(*Intent*)) call.

- *JNI call*, jni(mtd, obj). A *Java Native Interface* (JNI) call can be invoked by the Java layers (namely A, AF, and AR) to access a C/C++ method *mtd* in layer L using *obj* as object reference for the JNI call. For instance, the Activity Manager Service uses jni(`getCallingPid`, `null`) to get the process ID of the application that is currently invoking its functionality.

- *Socket call*, socket(id, m). A socket call is a direct invocation of a Linux socket. It is normally invoked from the AF and L layers. A bytestream m is delivered to the socket id. For instance, the Activity Manager Service can send a message M to the Zygote socket by means of the call socket(`"zygote"`, M). Direct invocation of socket calls by applications is strongly discouraged (cf. [2]) although not forbidden.

- *System call*, sys(id, *args*). A system call is used to directly invoke a Linux native kernel functionality. The AF and L layers are expected to directly invoke system calls through functions in the *bionic libc* library in order to provide services to the above applications. For example, in order to create a new process the Zygote library sends a sys(`fork`, `null`) call to the Linux kernel.

- *Function call*, func(id, *args*). A function call is an intra-component invocation that can be invoked in all layers. For example, in order to determine if a given component (represented by *pid* and *uid*) has a particular permission (i.e. $android.permission.INTERNET$), the Activity Manager Service can invoke the call func(`checkComponentPermission`, P) that belongs to the Package Manager Service. In this case $P = \{pid, uid, android.permission.INTERNET\}$.

- *Dynamic load call*, dl(id). It allows the A and AF layers to retrieve a pre-compiled library (residing in the L layer) identified by id. For instance, `GPSLocationProvider` loads the GPS libraries through the call dl(`"libgps.so"`).

Each call is invoked by a component and it is targeted to another component residing in the same or in other layers. Figure 1 shows the Android layers involved in the different types of call.

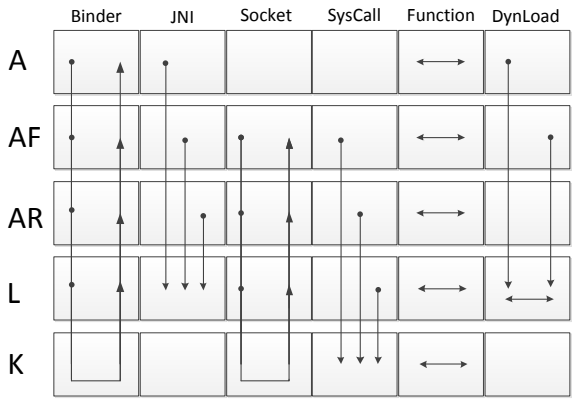

Figure 1: Potential sources (dots) and destination (arrowheads) of Android calls

We indicate with $X_\ell \xrightarrow{c} Y_{\ell'}$ the successful invocation of a call c having X and Y as source and the target components respectively. For instance, $app_{\mathrm{A}} \xrightarrow{\mathrm{binder}(\ldots)} TelMan_{\mathrm{AF}}$ denotes an invocation of the Telephony Manager service from a requesting application *app* by means of a Binder call. Similarly, $app_{\mathrm{A}} \xrightarrow{\mathrm{socket}(id,m)} Service_{\mathrm{AF}}$ means that $Service_{\mathrm{AF}}$ has received data m from app_{A} on socket id.
A *flow* is as a sequence of calls apt to perform an operation on the system. Common operations in an Android smartphone may be performing a phone call, access the media library, get the position of the mobile phone through the GPS, to name a few. For instance, the flow that gives the GPS position to an application *App* (cf. [1]) is: $App_{\mathrm{A}} \xrightarrow{\mathrm{binder}(\ldots)} LMS_{\mathrm{AF}} \xrightarrow{\mathrm{func}(\ldots)} GLP_{\mathrm{AF}} \xrightarrow{\mathrm{jni}(\ldots)} GLP_{\mathrm{L}} \xrightarrow{\mathrm{dl}(\ldots)} GL_{\mathrm{L}} \xrightarrow{\mathrm{sys}(\ldots)} KD_{\mathrm{K}}$, where LMS is the Location Manager Service, GLP is the GPS Location Provider (initially as AF service, then with its native implementation), GL is the GPS library, and KD is the corresponding Kernel Driver.

The concept of flow is key to reason about security-relevant aspects of the Android cross-layer architecture [1].Unfortunately, a systematic account of the legitimate flows is not available in the Android documentation [2, 1]. For instance, only three sample flows are provided in the official documentation. But these flows do not support a large part of common smartphone operations, which are therefore carried out through non-documented flows.
It must be noticed that the Android Security Framework (ASF) can enforce security checks on the individual calls, but it provide no support for checks encompassing an entire flow. This may affect security: an operation can be executed by a malicious component/applications by means of a slightly modified flow which cannot be possibly recognized as illegal by the ASF.
In order to give evidence of the security implications related to flows, in the following section we describe a vulnerability related to the launch of a new application in Android. The launch of a new application is based on a flow which is not formalized in the Android documentation. We show how a modified flow can be carried out by a malicious application, thereby witnessing the limitations of the ASF.

4. THE ZYGOTE VULNERABILITY

In Android, the launch of a new application requires the creation of a new process at layer K, the instantiation of a new DVM, and the binding of the process with the DVM. In this section we describe the standard flow implemented in Android for launching a new application. As show in [3] a malicious application can exploit the lack of control in the ASF to build a different flow that seriously affects the performance of the smartphone. Due to the lack of controls related to flows, the ASF has no means to detect and prevent the execution of the malicious flow.

4.1 Zygote: the standard flow

When an application is launched a `startActivity` request is sent to the *Activity Manager Service*, a part of the System Server, by means of an intent. The *Activity Manager Service* determines if the application has already an attached process at the Linux layer, the K layer, or if a new one is needed. The first case happens when an instance of the application has been previously started and it is currently executing in background; when this is the case the Activity Manager Service gets the corresponding process and brings back the application to foreground. In the second case, the Activity Manager Service calls `Process.start()`, a method of the static class `android.os.process`. This method executes a socket call connecting the Activity Manager Service to the the Zygote socket in order to send a command requesting the creation of a new process at the Linux layer. The Zygote socket is checked by a proper service called Zygote process that has the exclusive right to invoke the fork system call at the K layer. Thus, the command to create a new process is issued to the Zygote process through the Zygote socket.
The Zygote process gets the command from the Zygote socket and performs some security checks based on a built-in security policy. If the checks are passed, a JNI call to the function `ForkAndSpecialize` in the Zygote library is executed. The command sent to the Zygote socket includes a set of parameters. Some parameters are passed to the `ForkAndSpecialize` function. The most important parameter is the name of the class whose static `main` method will be invoked to specialize the child process. The Activity Manager Service uses a static class (namely `android.app.ActivityThread`) to specialize the new process with a standard DVM. In this class, a binding operation between the Linux process and an Android application is attempted. If no application is available for binding, the same class requires to kill its containing process through a `kill` system call to the Kernel.
If the spawning of the new process and the binding operation succeed, the Zygote process returns its child's PID to the Activity Manager Service.
The corresponding flow, depicted in Fig. 2, is:
$AL_{\mathrm{A}} \xrightarrow{\mathrm{binder}(\texttt{StartActivity}(\mathit{Intent}))} AM_{\mathrm{AF}} \xrightarrow{\mathrm{socket}(ZS_{\mathrm{K}},\mathit{ForkCmd})} ZP_{\mathrm{AR}} \xrightarrow{\mathrm{jni}(\mathit{FaS},\mathit{params})} ZL_{\mathrm{L}} \xrightarrow{\mathrm{sys}(\mathit{fork},\emptyset)} Kernel_{\mathrm{K}} [\xrightarrow{\mathrm{sys}(\mathit{kill},\mathit{self})} Kernel_{\mathrm{K}}]$, where the call in brackets is optional.
As stated above, the ASF does not check the entire flow.

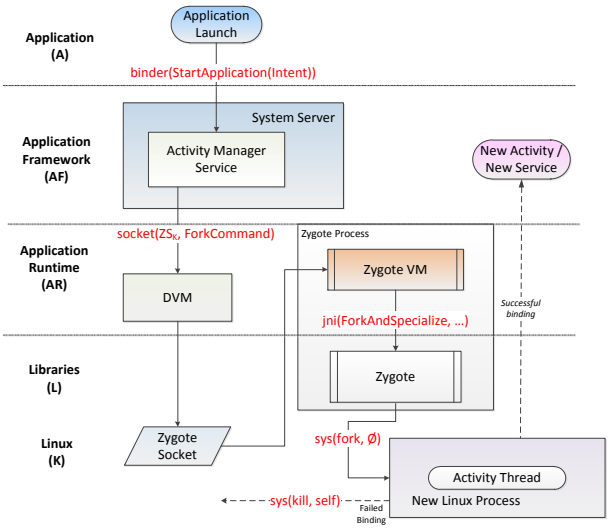

Figure 2: Standard flow for launching a new application in Android.

On the contrary, checks are only performed on single calls. In the specific case, partial security checks are made on the invocation of the socket($ZS_{\mathrm{K}}, ForkCmd$) on the *ForkCmd* parameters but no checks are made, for instance, on the identity of the invoking component.

4.2 Building a malicious flow

The Zygote socket is owned by `root` but has permissions `666` (i.e. `rw-rw-rw`). This means that any user process can read and write on it and hence send commands to the Zygote process. This choice is justified by the need of the process hosting the Activity Manager Service (whose UID is statically defined as `SYSTEM_UID`, it is not owned by root, nor it belongs to the root group) to request the spawning of new processes. However, this has the unintended effect to enable any process to ask for a fork. In terms of flows, this means that the original flow can be modified and the corresponding socket call can be made by any active component in the system, instead of the Activity Manager Service as expected.
However, to avoid misuse, the security policy enforced by the Zygote process restricts the execution of the command received (i.e. `ForkCmd`) on the Zygote socket.
The policy prevents from (a) issuing the command that specifies a UID and a GID for the creation of new processes if the caller is not root nor the Activity Manager Service, (b) creating a child process with more capabilities than its parent, and (c) enabling debug-mode flags and specifying `rlimit` bounds if requestor is not root or the System is not in "factory test mode".

Only a few checks are performed on the (static) class used to customize the child process: 1) whether the class contains a static `main()` method and 2) whether it belongs to the System package, which is the only one accepted by the Dalvik System Class loader. Unfortunately, these security checks do not include a proper control of the identity (i.e. UID) of the caller. Therefore each Linux process (and hence the associated Android application or service) can send fork commands to the Zygote socket as long as a valid static class is provided.
As show in [3], by using the System static class `com.android.internal.util.WithFramework` it is possible to force the Zygote process to fork, generating a dummy process which is kept alive at the Linux layer. Such class does not perform any binding operation with an Android application, thus Android mechanism that removes unbound new processes (as the `android.app.ActivityThread` class does) is not triggered.
In this way, all the security policies applied by the Zygote process are by-passed, leaving a persistent Linux process which occupies memory resources in the device.
The resulting malicious flow, depicted in Fig. 3, is:
$MalApp_{\mathrm{A}} \xrightarrow{\mathrm{socket}(ZS_{\mathrm{K}}, MalForkCmd)} ZP_{\mathrm{AR}} \xrightarrow{\mathrm{jni}(FaS, MalParams)} ZL_{\mathrm{L}} \xrightarrow{\mathrm{sys}(fork, \emptyset))} Kernel_{\mathrm{K}}$.

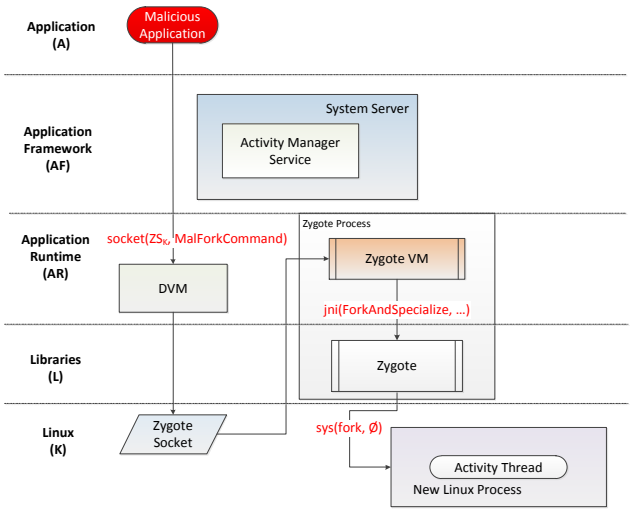

Figure 3: A malicious forking flow.

Note that this flow completely by-passes the System Server (and the whole AF layer) which is responsible for managing the creation and destruction of new processes and applications.

4.3 Impact of the malicious flow

By repeating the malicious flow the Zygote socket can be flooded by fork requests and this quickly leads to the exhaustion of the resources. The ASF proves to be weak against both the single and the multiple execution of the malicious flow. In fact, no Android layer can detect the generation of the dummy processes and then to intervene.
On the other hand, the creation of processes at the Linux layer is legal since the fork operation is executed in kernel mode. Thus, the ASF is unable to recognize such behavior as malicious.
As soon as the dummy processes consume all the available resources, a safety mechanism reboots the device. This is to no avail. In fact, by forcing the execution of the malicious flow during boot-strapping, it is possible to lock the device into an endless boot-loop, thereby locking the use of the device.

Notice that to mount the attack, the malicious application does not require any special permission, therefore it looks harmless to the user upon installation. The attack has been tested on a large number of emulated and real devices with different Android versions and all of them proved vulnerable. Further information on the testbed and results can be found in [3].

4.4 Patching the ASF

The malicious flow differs from the standard one in two parts:

1. The socket call to the Zygote socket is invoked by the malicious application, i.e. $MalApp_{\mathrm{A}} \xrightarrow{\mathrm{socket}(ZS_{\mathrm{K}}, MalForkCmd)} ZP_{\mathrm{AR}}$. In the standard flow the Activity Manager Service is instead activated by a request from the A layer, i.e. $AL_{\mathrm{A}} \xrightarrow{\mathrm{binder}(\texttt{StartActivity}(Intent))} AM_{\mathrm{AF}} \xrightarrow{\mathrm{socket}(ZS_{\mathrm{K}}, ForkCmd)} ZP_{\mathrm{AR}}$.
2. The optional system call for killing the unbound process is avoided ($Kernel_{\mathrm{K}} \xrightarrow{\mathrm{sys}(kill, self)} Kernel_{\mathrm{K}}$).

We have built up a patch for the ASF that prevents malicious applications from invoking fork requests. Our patch operates at layer K only by restricting permissions on the Zygote socket. The patch takes advantage of the sandboxing mechanism of Android, that forces each Android application (except in very few cases) and service to execute as a separate Linux user. The System Server is executed by a Linux user with $GID =$ `system` in all Android versions. Thus, we changed the Zygote socket group from `root` to `system` and reduced the access permissions from `666` to `660`.
In this way, only socket calls coming from the System Server are successfully executed, while others are discarded by the Linux native permission system.
The proposed patch—which has been adopted by the Android Security Team and it has been applied in version 4.0.4—avoids the execution of the malicious flow by blocking the possibility for an application to successfully execute a socket call to the Zygote socket, allowing only legal calls from the System Server. The patch works under the (reasonable) assumption that the System Server is trusted. In this way, we can also assume that the patch is robust against the second modification of the optional kill system call, since the System Server is expected to use the standard class for specializing the child process.

The Zygote vulnerability witnesses the difficulties in assessing the security of flows in Android. The ASF is unable to relate different calls and tell apart malicious and legitimate flows. To this aim, our patch is strong against the malicious flow presented in 4.2 but it does not provide Android with the ability to analyze flows and thus it could be of no help when it comes to countering other potentially malicious flows. Thus, we argue that the security assessment of flows is an open security issue of Android and that the ASF should be given the ability to analyze flows.

5. TESTING THE ASF ON CALLS

The Zygote vulnerability is basically due to a lack of control on the identity of the components invoking calls targeted to the K layer that are normally expected to be executed by services in the AF layer. Successful invocation of such calls by malicious applications instead of legal services in the Android stack may potentially lead to other vulnerabilities as the one described in Sect. 4.
As a first step towards a systematic analysis of flows in Android, we carried out a empirical assessment on the calls to the K layer (i.e. binder, socket and system calls, cf. Fig. 1) with the objective to verify whether the ASF is able to recognize whether that an execution of a call, normally executed by legal services at AF layer, is invoked by a malicious application.
To this end we modified both Android and the Linux kernel in such a way to capture all binder, system, and socket calls that the Android services in the AF layer normally invoke during their execution. Then, we implemented a tester application that invokes the same calls (i.e. with the same parameters) from the A layer. This allowed us to empirically assess whether the ASF is able to discriminate between the origin of calls, and, in case, to intervene.
As mentioned in Sect. 2, calls from the A and AF layers should be mediated by the AR by the means of the DVM.
We set up a testing scenario into two steps: we implemented 1) a *monitoring kernel module* (MKM) able to intercept and log system, socket and binder calls and 2) a tester application that executes (from the A layer) the calls intercepted by the MKM.
Both the kernel module and the tester application have been developed and tested in an emulated Android device with a Linux kernel *goldfish* v. 2.6.29, compiled with *arm-eabi-4.2.1 toolchain*, and Android v. 2.3.3 and v. 3.2.

5.1 Building the monitoring Kernel module

Since the ability to load modules is disabled in the Linux kernel of Android, we modified the kernel to enable this feature and recompiled it for a generic ARM architecture. We then pushed the modified kernel module on the device and installed it via the `adb` shell, using the `insmod` command.
Since the routines to be executed in response to Linux system calls are declared in the `sys_call_table` structure, our modified kernel module substitutes each entry in the table with a custom routine. Such a routine gets the calling thread name and process pid (using the Linux macro `current`) as well as the optional parameters passed to the system call. At the end of the custom routine, the actual system call is executed. We use `systemcalls.h` for system calls prototypes and `unistd.h` for system calls numbers.
The MKM offers two ways of logging: 1) writing on an existing system log or 2) writing on an ad-hoc char device. In the former case, the MKM writes down logs on `dmesg` kernel ring buffer. In the latter case, the MKM uses a custom driver for a char device which supports open, close, read, and write operations. During installation, the MKM creates a new char device in `/dev` called `sysCon` and attaches the device to its driver. Once a call is executed, an entry is written on the device by the MKM. In user space, applications are able to connect to `sysCon` and retrieve data stored in it.

Our tester application, called `SysCallTester`, simply reads the data written on the `sysCon` device and re-invokes the

system calls with the same parameters as the original one. This application can also write parsed data in a text file for permanent storage.

5.2 Testbed

We tested the MKM and `SysCallTester` on Android emulators v. 2.3.3 (API 10) and v. 3.2 (API 13) using our modified Linux kernel. On v. 2.3.3 we have developed a custom `rc.module` script which is executed as a service in the `init.rc`. The script installs the kernel module and acquires the data written on `dmesg` every 30 seconds for 5 minutes and stores them in a text file. The modification of the `init` file leads to the creation of a custom `ramdisk.img` file for the emulated device.

During the test execution, we repeatedly performed general-purpose operations on the emulated device in order to force the AF services to execute calls. Such operations include the launch of a new application, installation/uninstallation of applications, browsing, email-related operations and execution of random applications.

We logged a subset of the most representative system calls. This set includes core system calls (e.g. for I/O and process management), socket calls and binder calls (which rely on the `ioctl` system call). It is worth noticing that the whole set can be easily extended, adding the prototype of the new system call in the MKM.

The automatic launch of the tester application has been automated using a `BroadcastReceiver` which intercepts the `BOOT_COMPLETED` system message.

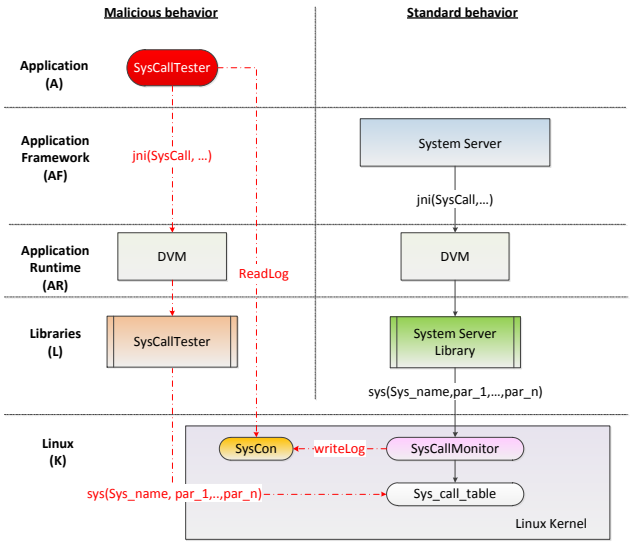

Figure 4: The testbed

In our tests, `SysCallTester` and the MKM are executed at the same time:

`SysCallMonitor` intercepts the system calls made by the *System Server*, logging the data on its char device `sysCon` as shown in Fig. 4. `SysCallTester` keeps monitoring the log from `sysCon` and, as soon as a new system call is logged on `sysCon`, it invokes its C++ library in order to re-execute the system call, including the proper parameters.

5.3 Experimental results

Our experiments allowed us to empirically identify which components in the AF layer invoke which system calls. The result of our analysis is summarized in Table 1.

Table 1: System calls invoked by services in the AF layer.

AF service	SysCalls
Alarm Manager	`getpid`, `ioctl`, `open`
Activity Manager	`close`, `getpid`, `gettid`, `ioctl`, `lseek`, `mkdir`, `open`, `prctl`, `read`, `write`
Audio Service	-
BatteryStats	`close`, `exit`, `gettid`, `open`
GpsLocationProvider	`getpid`, `ioctl`
Location Manager Service	`getpid`, `ioctl`, `lseek`, `open`, `read`
Package Manager	`close`, `getpid`, `gettid`, `ioctl`, `lstat64`, `open`, `sendmsg`, `write`
Power Manager Service	`getpid`, `ioctl`, `open`, `read`, `write`
ServerThread	`close`, `connect`, `getpid`, `gettid`, `ioctl`, `lseek`, `lstat64`, `open`, `prctl`, `read`, `recvmsg`, `sendmsg`, `sendto`, `socket`, `write`
ThrottleService	`close`, `exit_group`, `getpid`, `gettid`, `ioctl`, `open`, `prctl`, `read`, `sendmsg`, `write`
VoldConnector	`getpid`, `gettid`, `ioctl`, `open`, `recvmsg`, `write`
Window Manager	`close`, `getpid`, `gettid`, `ioctl`, `open`, `read`, `write`

Moreover, in order to check whether the ASF is able to discriminate between different callers of the same instance of the call, we have re-executed through `SysCallTester` (invoked 50 times) the following system calls as soon as they were executed by the legal service in the AF layer: `bind`, `close`, `connect`, `exit_group`, `exit`, `getpid`, `gettid`, `kill`, `ioctl`, `lseek`, `lstat64`, `mkdir`, `open`, `prctl`, `read`, `recvfrom`, `recvmsg`, `sendmsg`, `sendto`, `socket`, `write`. `SysCallTester` was able to re-execute properly the 85% of the intercepted system calls (18 out of 21), both on Android v. 2.3.3 and v. 3.2. Only three calls systematically fail (i.e. `bind`, `kill`, and `sendto`). This is due to unaccepted parameters: `bind` fails because the targeted socket is already bound, `kill` fails because it is not possible for a normal user to kill another process except itself, and `sendto` cannot be executed because another endpoint is already connected. Our experiments indicate that little control is exercised among the *Android layers* (A, AF, AR and L) and the Linux layer (K) regarding system, socket and binder calls. The System Server is, from a kernel point of view, a normal Linux user since it has no root privileges. For this reason, the Linux kernel is not able to discriminate System Server calls from those invoked by another Linux user (e.g. a rogue application) let alone to counter them. The ASF is also apparently unable to discriminate the callers of a call, thus potentially permitting malicious flows as it is the case of the *Zygote* vulnerability).

6. RELATED WORK

Android security is an emerging research field, in particular due to the fact that user personal data are managed on a device that is generally kept continuously connected to the Internet. Current literature is mainly devoted to propose solution for assessing/improving the privacy/security of the end-user by means of *i*) static analysis on the compliance of Android applications against expected security properties [8], [9], [10], *ii*) enhancements for ASF (and related security policy) [5, 6], and *iii*) detection of vulnerabilities and security threats [3, 4].

Independently from the final aim, some proposals directly or indirectly deal with Android calls. For example, approaches based on static analysis indirectly deal with cross-layer calls. In [8], a horizontal study of Android applications aimed at discovering stealing of personal data is performed. Besides, in [9] a black-box analysis tool (Stowaway) for inferring overprivilege in compiled Android applications is proposed, while in [10] a tool (Scandroid) for automatically reasoning about the security of Android applications is discussed. In particular, Scandroid takes into account interactions between applications. Enhancements of native ASF are mostly aimed at improving the management and the granularity of the native Android permission system. In particular, in [12] a policy enforcement framework (Apex) allows the end user to selectively grant permissions to the applications. In [13] a modified infrastructure (SAINT) aimed at managing install-time permissions assignment is proposed. [4] proposes a security framework (XManDroid) that extends the native monitoring mechanism of Android to detect privilege escalation attacks. However, none of such solutions, albeit dealing with Android calls, takes into consideration interplay between the Android stack and the Linux kernel, focusing attention only on invocations related to the Android layers. Furthermore, no proposal deal with the idea of flow nor it assesses the impact of cross-layer flows on the security of Android. To the best of our knowledge, our work is the first attempt to investigate correlations and security issues related to the interplay in the whole Android architecture.

7. CONCLUSIONS

In this paper, we have discussed security issues related to interplay in Android platforms and provided a preliminary assessment of the security implications related to the cross-layer interplay of the Android components. We have showed that attacks to the security of Android may be driven by malicious applications and that the ASF as well as the native security mechanisms at the Linux layer may be not sufficient to discriminate between the caller of an invocation. Such a scenario may lead to vulnerabilities whose exploitation by malicious applications may go undetected as it is the case for the Zygote vulnerability.

To support our observations we have developed 1) a kernel module that logs system calls invoked by AF layer and 2) a tester application capable to read the logs and re-execute successfully the tracked calls. Our experiments indicate that little control is exercised among the Android and the Linux layers, thereby indicating that the attack surface of the Android platform is wider than expected.

8. REFERENCES

[1] Android Development Team. Anatomy and physiology of an android.

[2] Android Project Documentation. Android Application Security. http://source.android.com/tech/security/index.html, May 2012.

[3] A. Armando, A. Merlo, M. Migliardi, and L. Verderame. Would you mind forking this process? A denial of service attack on Android (and some countermeasures). In *Proc. of the 27th IFIP International Information Security and Privacy Conference (SEC 2012), IFIP Advances in Information and Communication Technology, 376*, pages 13–24. Springer, 2012.

[4] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Technische Univ. Darmstadt, Apr 2011.

[5] I. Burguera, U. Zurutuza, and S. Nadjm-Therani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM'11)*, 2011.

[6] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[7] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In M. Burmester, G. Tsudik, S. Magliveras, and I. Ilic, editors, *Information Security*, volume 6531 of *LNCS*, pages 346–360. 2011.

[8] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.

[9] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, 2011.

[10] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. Scandroid: Automated security certification of android applications.

[11] G. Group. *Press Release*, November 2011. Available at http://www.gartner.com/it/page.jsp?id=1848514.

[12] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 328–332, New York, NY, USA, 2010. ACM.

[13] M. Ongtang, S. Mclaughlin, W. Enck, and P. Mcdaniel. Semantically rich application-centric security in android. In *In ACSAC '09: Annual Computer Security Applications Conference*, 2009.